GRAPHQL

REST HAS SOME PROBLEMS

- Over Fetching
- Under Fetching

WHY THESE ‘PROBLEMS’ MATTER

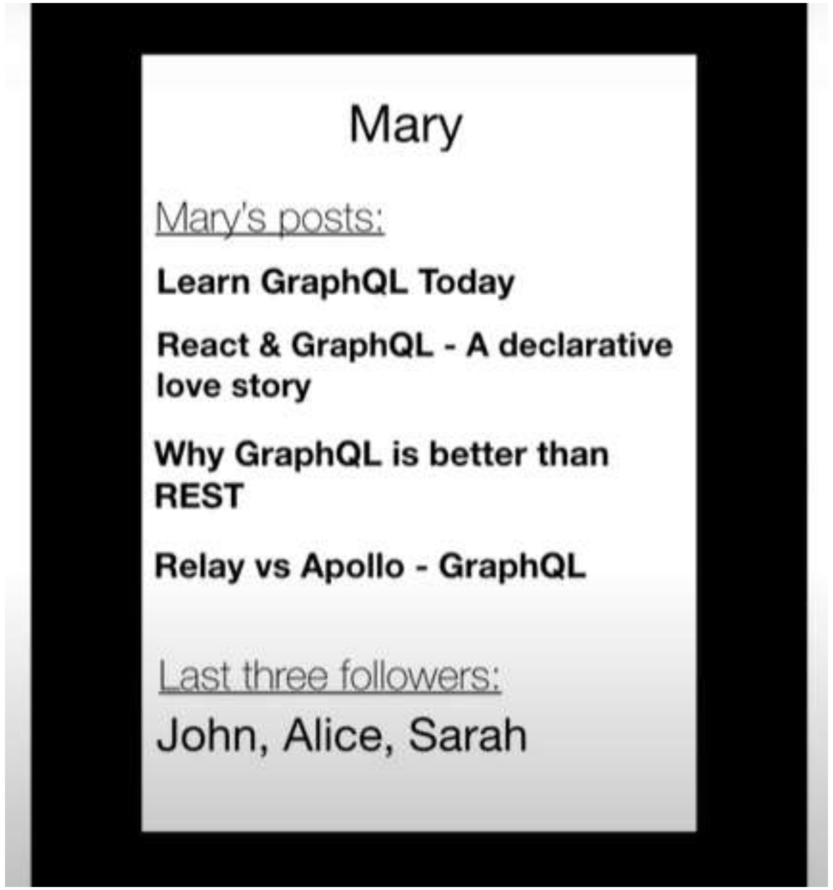
- Increased mobile clients (smart phones and smart devices) requires efficient data loading
- Client Heterogeneity (Example: Admin client vs Order Page -> accessing list of customers-orders)
- Faster development and deployment and rapid feature updates (hmm….grain of salt)

A LITTLE BIT OF HISTORY

- A Facebook (Meta) contribution
- Facebook started using it internally since 2012
- Made it public in 2015 (in a React.js Conference) and open sourced
- Lot of major companies have migrated their endpoints (or created new ones) to support GraphQL

GRAPHQL VS REST (BLOG APP)

- Print User Information
- List of Posts
- List of latest 3 followers
- Lets assume that the data is normalized
  - i.e. there are separate tables for
    1. User
    2. Posts
    3. Followers



Mary

Mary's posts:

Learn GraphQL Today

React & GraphQL - A declarative love story

Why GraphQL is better than REST

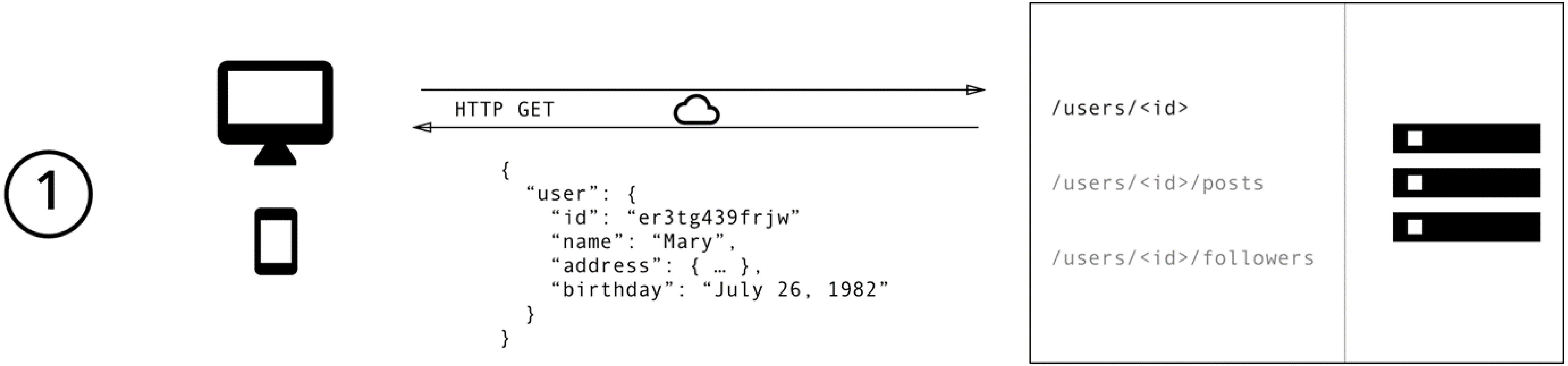
Relay vs Apollo - GraphQL

Last three followers:

John, Alice, Sarah

GRAPHQL VS REST (BLOG APP)

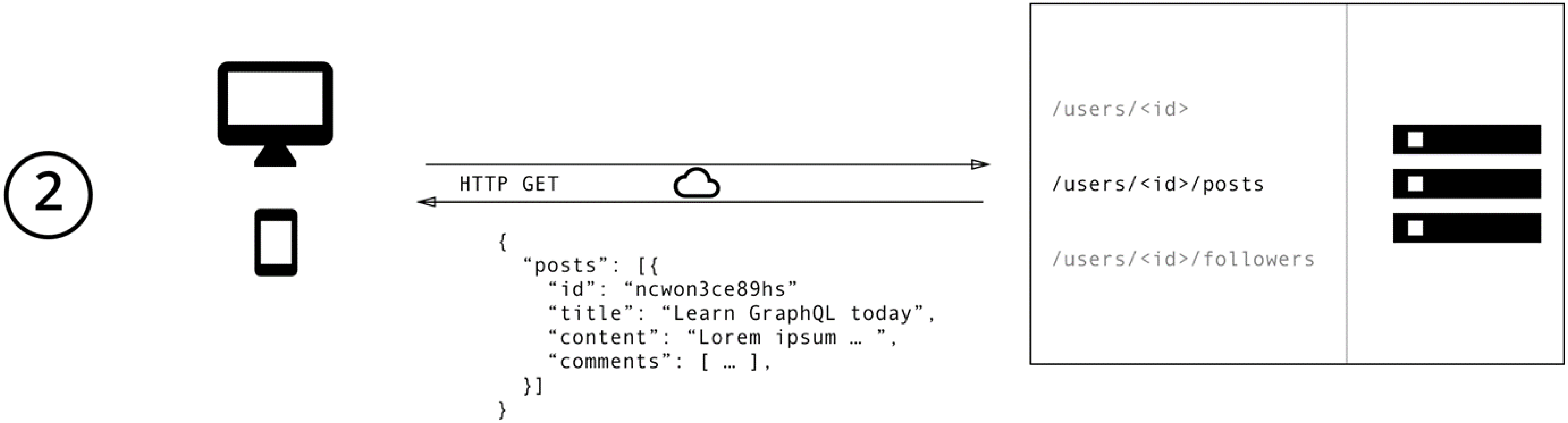
- First Get The User Information



```
{
  “user”: {
    “id”: “er3tg439frjw”
    “name”: “Mary”,
    “address”: { … },
    “birthday”: “July 26, 1982”
  }
}
```

GRAPHQL VS REST (BLOG APP)

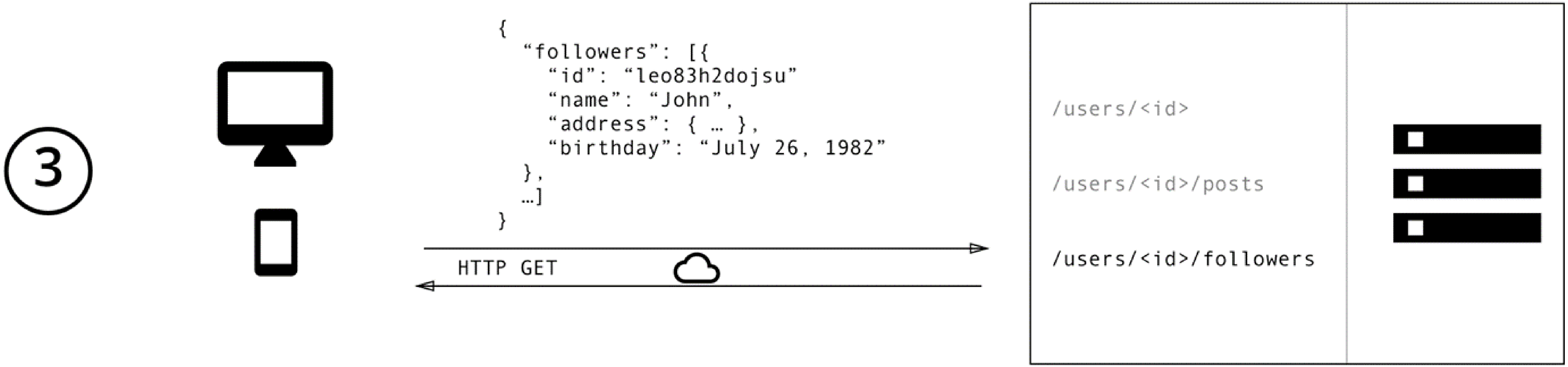
- Second Get The Posts Information



```
{
  “posts”: [{
    “id”: “ncwon3ce89hs”
    “title”: “Learn GraphQL today”,
    “content”: “Lorem ipsum … ”,
    “comments”: [ … ],
  }]
}
```

GRAPHQL VS REST (BLOG APP)

- Finally Get The Followers Information



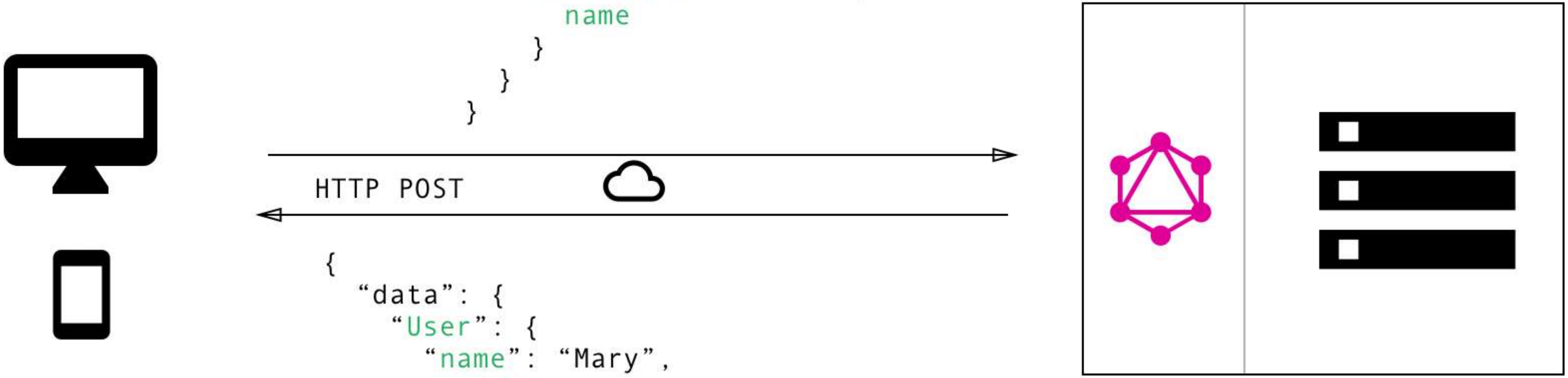
```
{
  “followers”: [{
    “id”: “leo83h2dojsu”
    “name”: “John”,
    “address”: { … },
    “birthday”: “July 26, 1982”
  },
  …]
}
```

HOW DOES THE SAME THING LOOKS IN GRAPHQL?

- Lets see!

```
query {
  User(id: “er3tg439frjw”) {
    name
    posts {
      title
    }
    followers(last: 3) {
      name
    }
  }
}
```

Specify Only What you need



```
{
  “data”: {
    “User”: {
      “name”: “Mary”,
      “posts”: [
        { title: “Learn GraphQL today” }
      ],
      “followers”: [
        { name: “John” },
        { name: “Alice” },
        { name: “Sarah” },
      ]
    }
  }
}
```

That’s exactly what you get

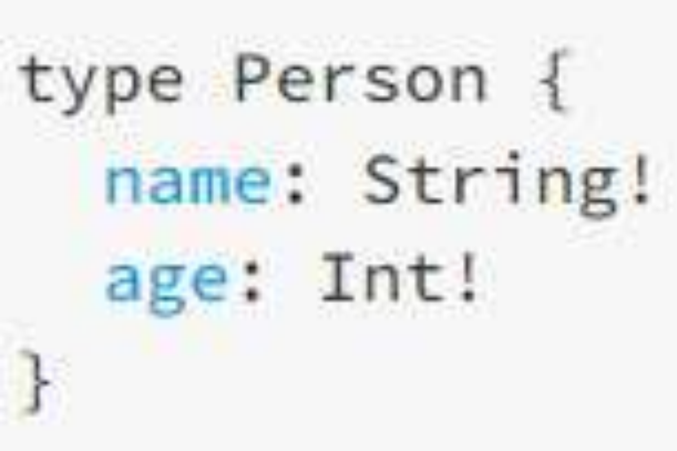
ANOTHER MAJOR BENEFIT: TYPES & SCHEMA

- GraphQL uses a strong type system to define capabilities of the API
  - Resource hierarchy as Type hierarchy
- The schema defines the contract between the client and the server
- Once there is an agreed upon schema
  - The front-end and back-end development can proceed independently

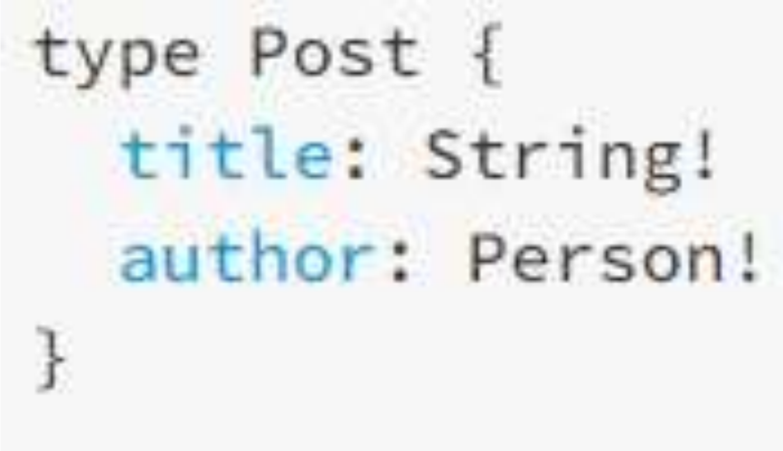
GRAPHQL CORE CONCEPTS

THE SCHEMA DEFINITION LANGUAGE (SDL)

- SDL for simple types
- Bang indicates ‘required’



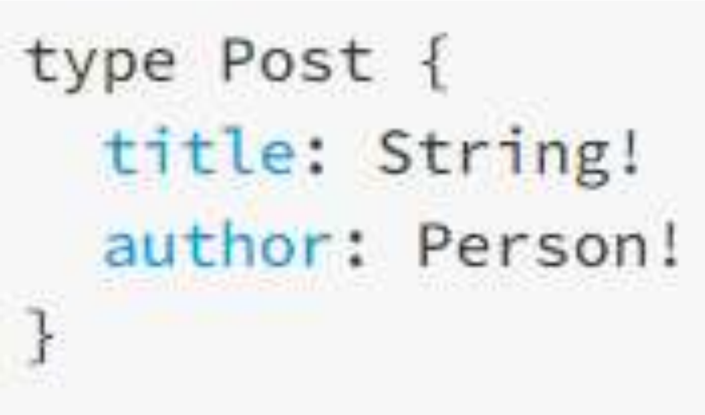
```
type Person {
  name: String!
  age: Int!
}
```



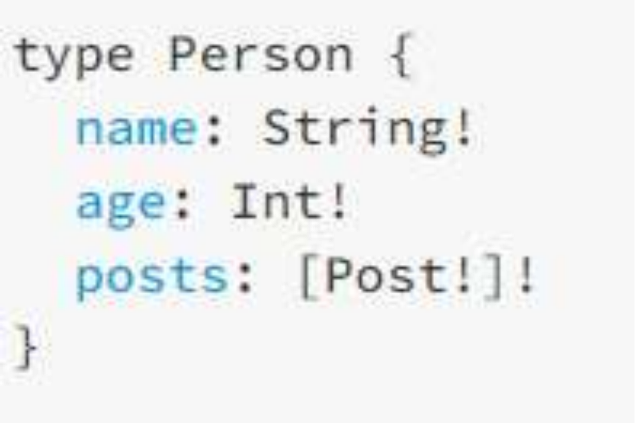
```
type Post {
  title: String!
  author: Person!
}
```

THE SCHEMA DEFINITION LANGUAGE (SDL)

- We can easily add relationships between types (similar to OOP)
- Lets see how we can add one-to-many between person and posts
- [ ] indicates collection



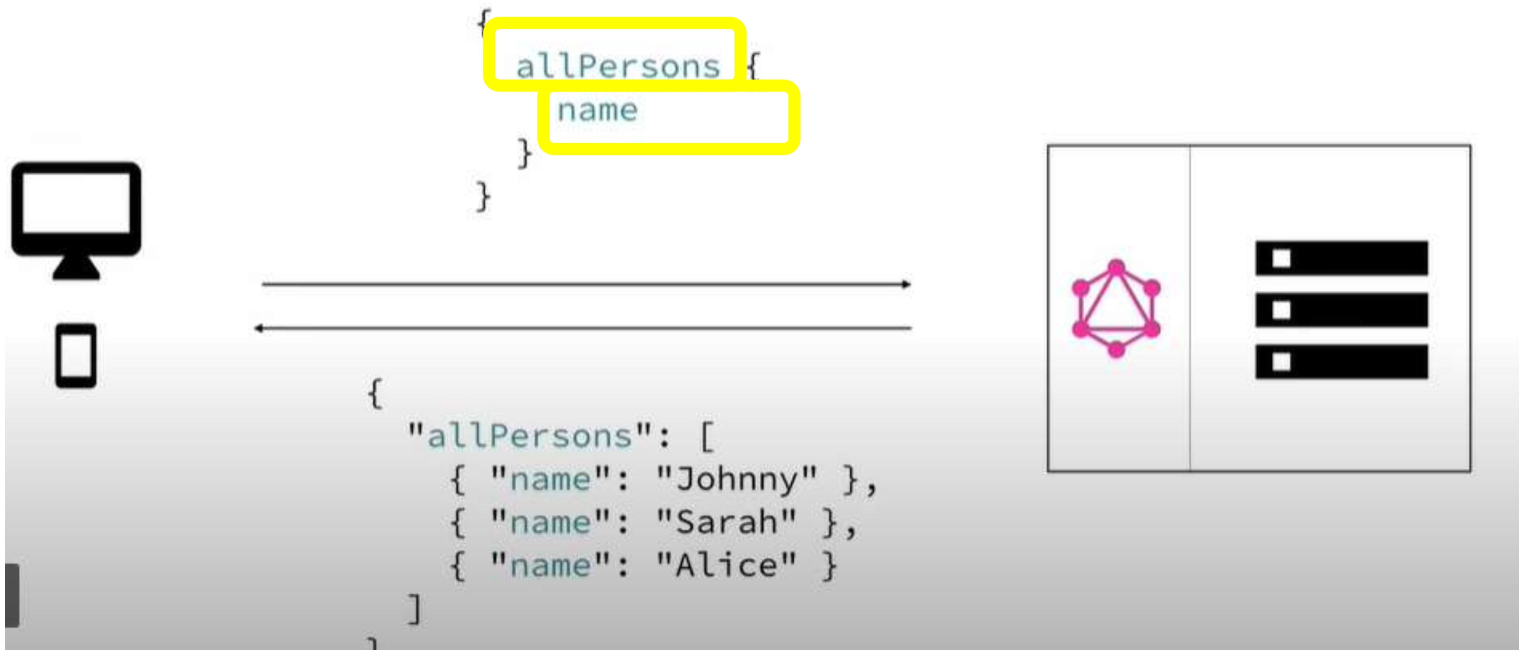
```
type Post {
  title: String!
  author: Person!
}
```



```
type Person {
  name: String!
  age: Int!
  posts: [Post!]!
}
```

BASIC GRAPHQL QUERIES

- Root-field
- Payload

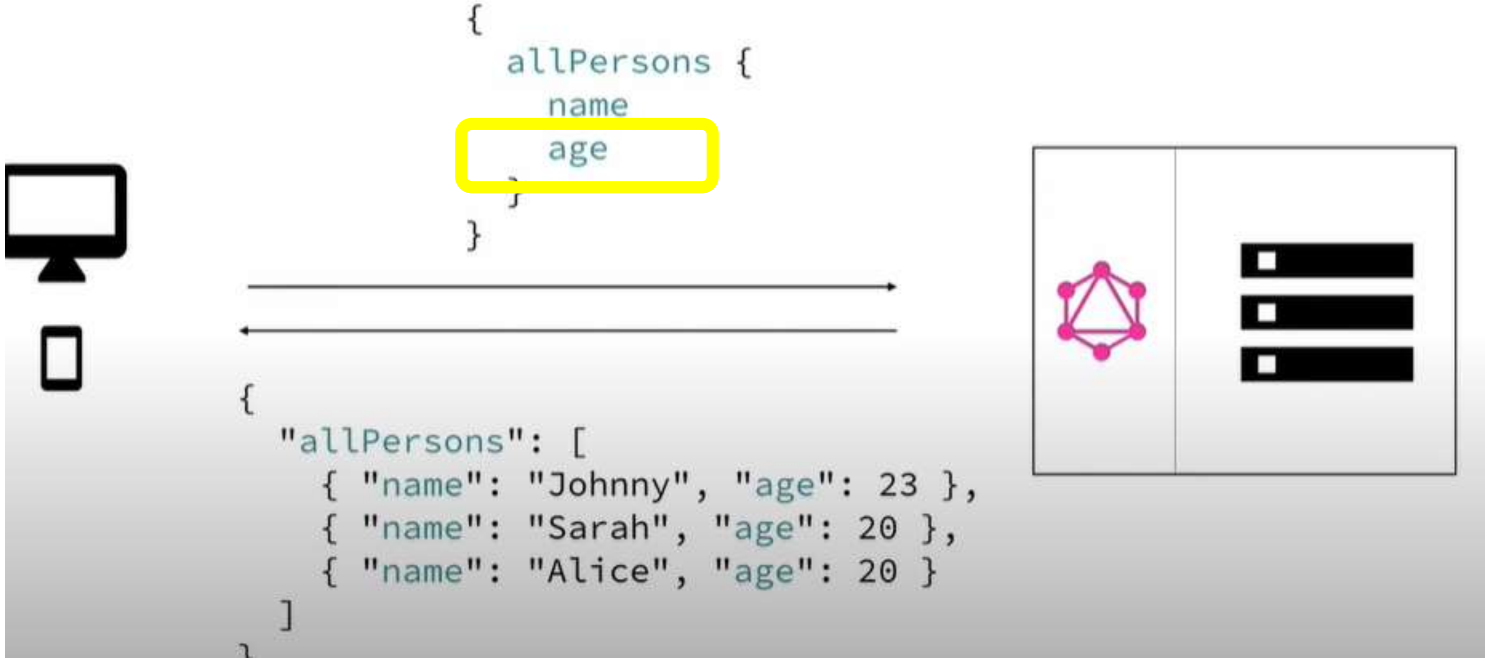


```
{
  allPersons {
    name
  }
}
```

```
{
  "allPersons": [
    { "name": "Johnny" },
    { "name": "Sarah" },
    { "name": "Alice" }
  ]
}
```

BASIC GRAPHQL QUERIES

- **Root field**
- **Payload** (modifying the payload slightly gives us more data)

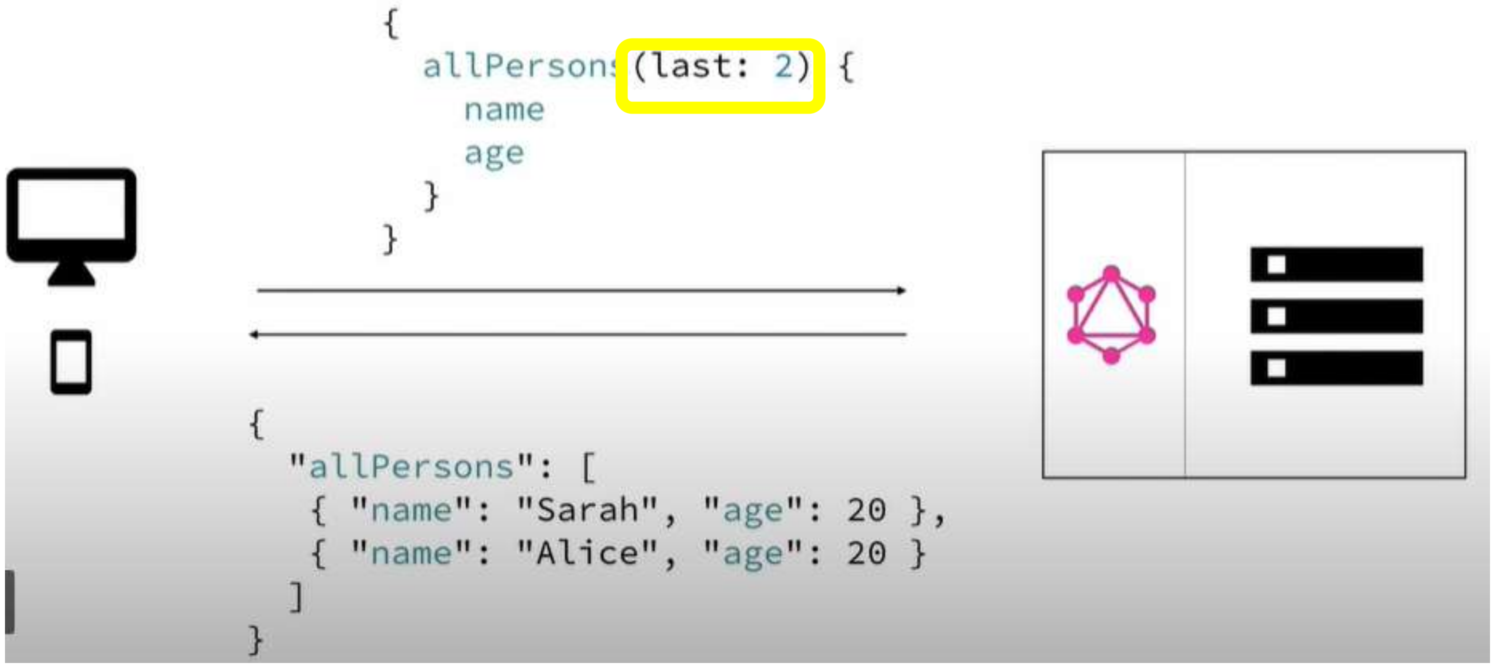


```
{
  allPersons {
    name
    age
  }
}
```

```
{
  "allPersons": [
    { "name": "Johnny", "age": 23 },
    { "name": "Sarah", "age": 20 },
    { "name": "Alice", "age": 20 }
  ]
}
```

BASIC GRAPHQL QUERIES

- Queries can accept **parameters**
- We can design parameters as we wish and support it in the backend

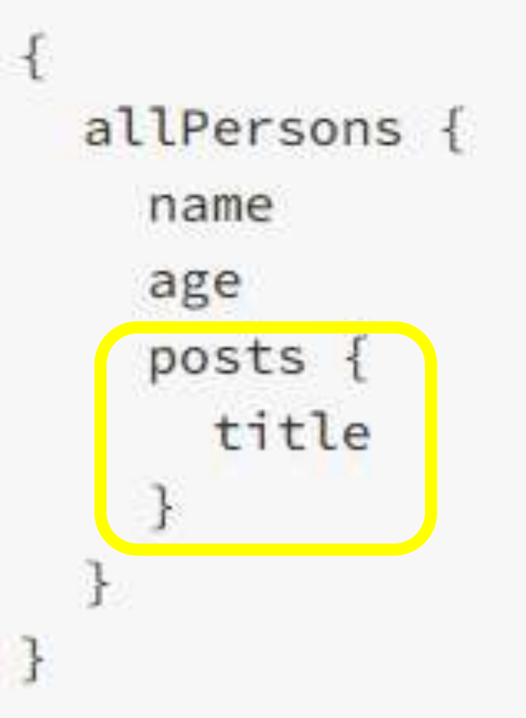


```
{
  allPersons(last: 2) {
    name
    age
  }
}
```

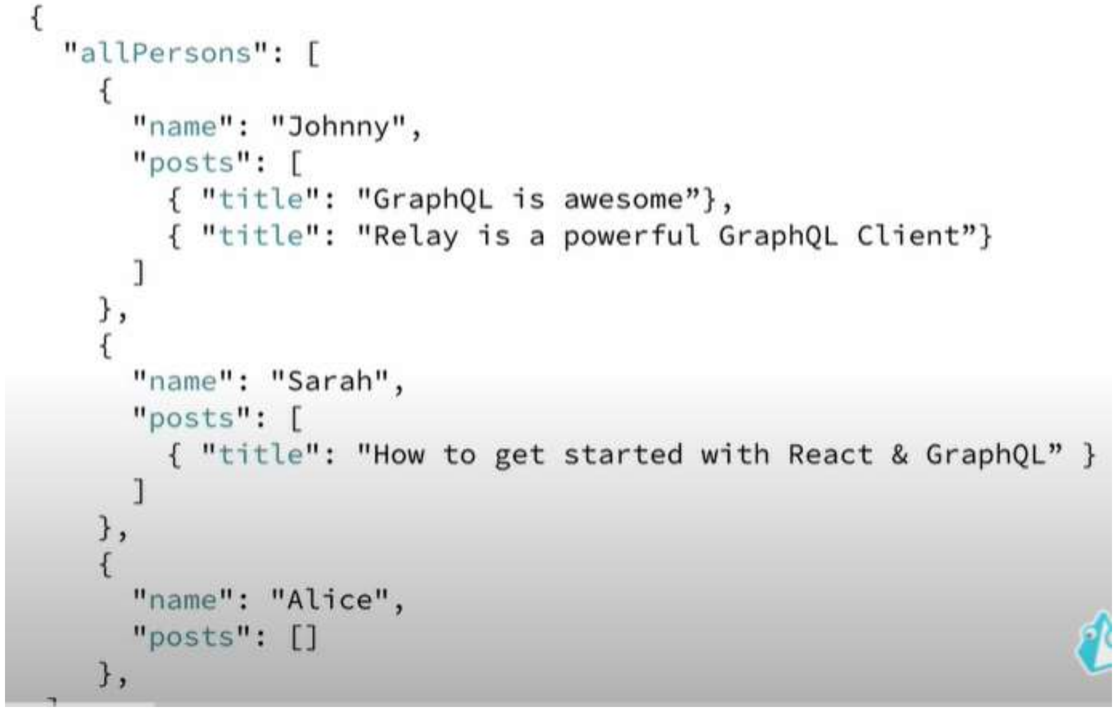
```
{
  "allPersons": [
   { "name": "Sarah", "age": 20 },
   { "name": "Alice", "age": 20 }
  ]
}
```

BASIC GRAPHQL QUERIES

- The beauty of GraphQL is the ability to support **nested Queries**
- Remember our person+posts schema



```
{
  allPersons {
    name
    age
    posts {
      title
    }
  }
}
```



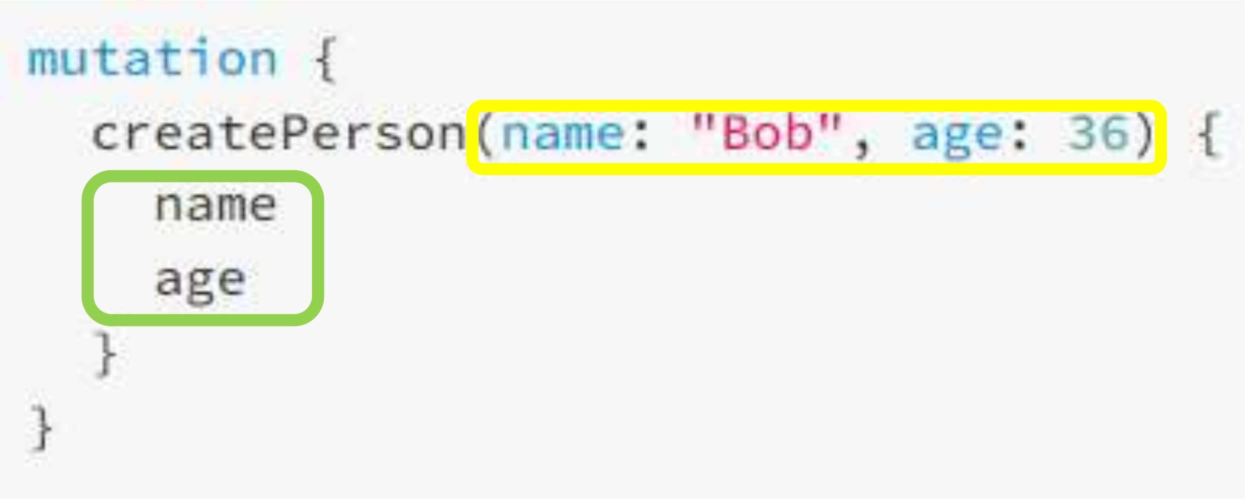
```
{
  "allPersons": [
    {
      "name": "Johnny",
      "posts": [
        { "title": "GraphQL is awesome”},
        { "title": "Relay is a powerful GraphQL Client”}
      ]
    },
    {
      "name": "Sarah",
      "posts": [
        { "title": "How to get started with React & GraphQL” }
      ]
    },
    {
      "name": "Alice",
      "posts": []
    },
```

CHANGING DATA (MUTATIONS)

1. Creation of new data
2. Updating existing data (both full and partial updates)
3. Deletion of data

MUTATIONS

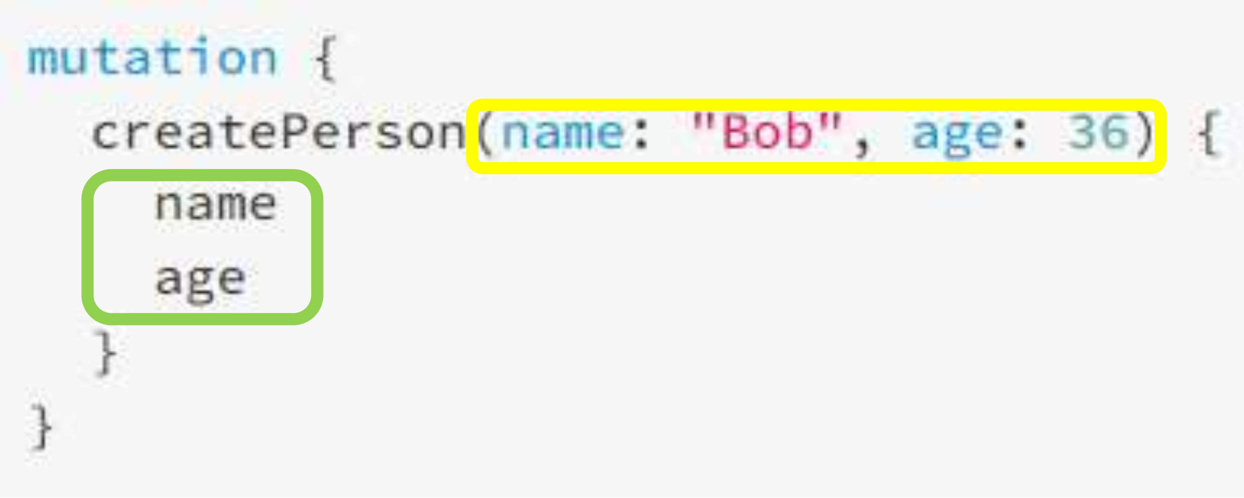
- Same syntactic structure as queries, but always starts with the mutation keyword
- Example of a **createperson** mutation



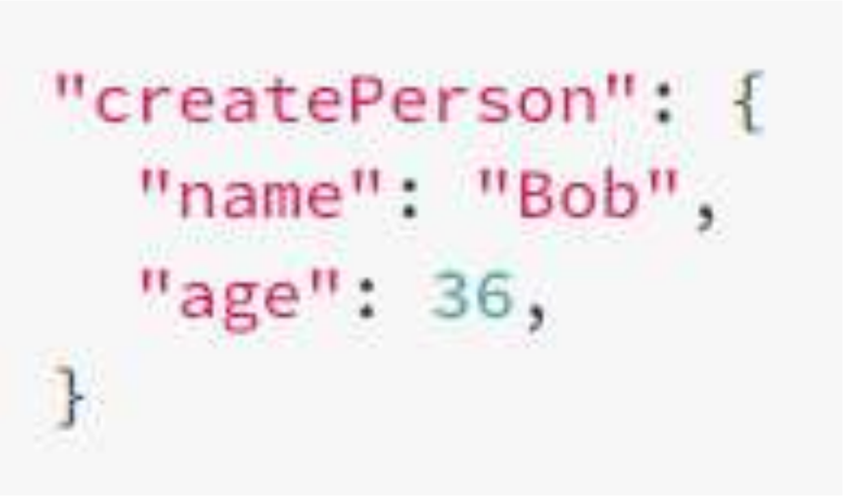
```
mutation {
  createPerson(name: "Bob", age: 36) {
    name
    age
  }
}
```

MUTATIONS

- Same syntactic structure as queries, but always starts with the mutation keyword
- Example of a **createperson** mutation



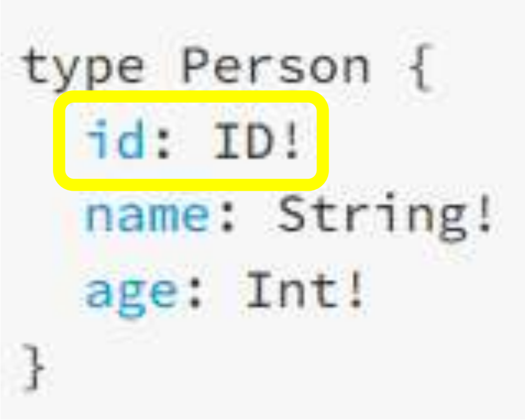
```
mutation {
  createPerson(name: "Bob", age: 36) {
    name
    age
  }
}
```



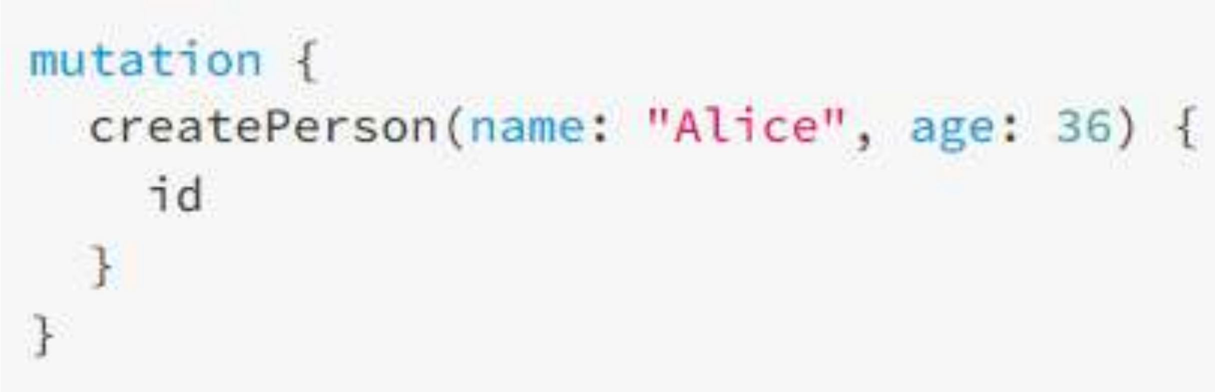
```
"createPerson": {
  "name": "Bob",
  "age": 36,
}
```

MUTATIONS

- One common pattern is to use the ID GraphQL type for uniqueIDs



```
type Person {
  id: ID!
  name: String!
  age: Int!
}
```



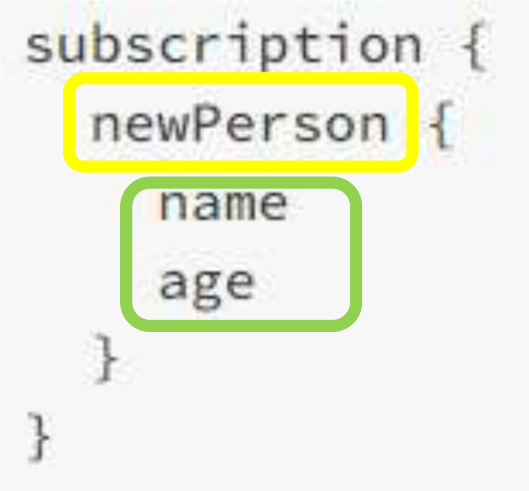
```
mutation {
  createPerson(name: "Alice", age: 36) {
    id
  }
}
```

MUTATIONS

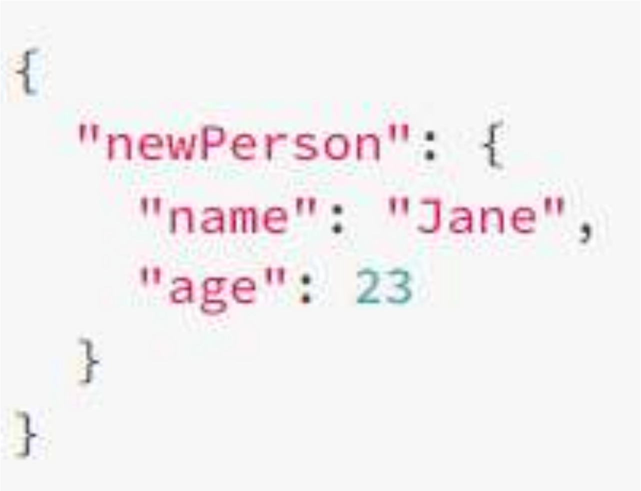
- Update Mutations does not require anything special
- You just pass in ID as one of the params, along with the params that needs to be updated

SUBSCRIPTIONS

- Another notable advantage of GraphQL is support to streaming Data
- Subscriptions represent a stream of data sent over to the client
- You subscribe to events and when that event happens the data you asked for is sent over



```
subscription {
  newPerson {
    name
    age
  }
}
```

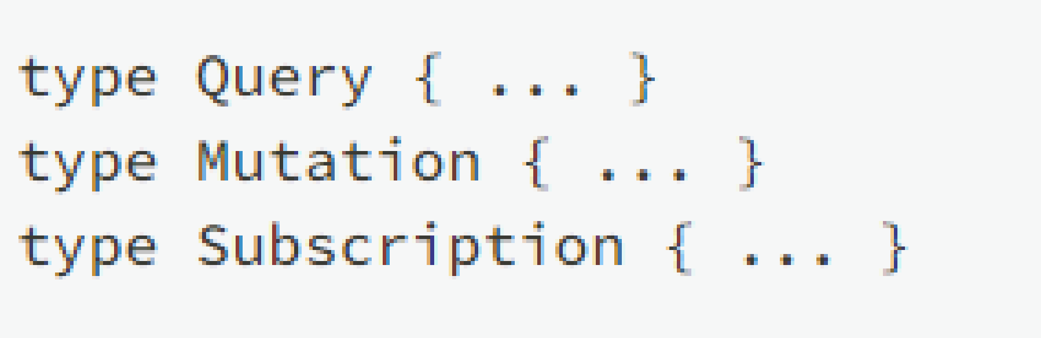


```
{
  "newPerson": {
    "name": "Jane",
    "age": 23
  }
}
```

Note: Subscription is out of scope for us. We don't expect you to stream data. You are free to explore more on your own.

LETS REVISIT SCHEMA

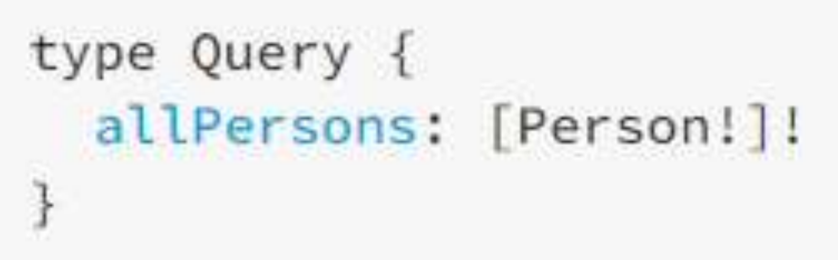
- Schema is simply a collection of GraphQL Types
- However (in convention) we prefer some typical root types, especially for APIs
- These are the entry points (Query, Mutation & Subscription).
- When you don’t specify anything it defaults to Query



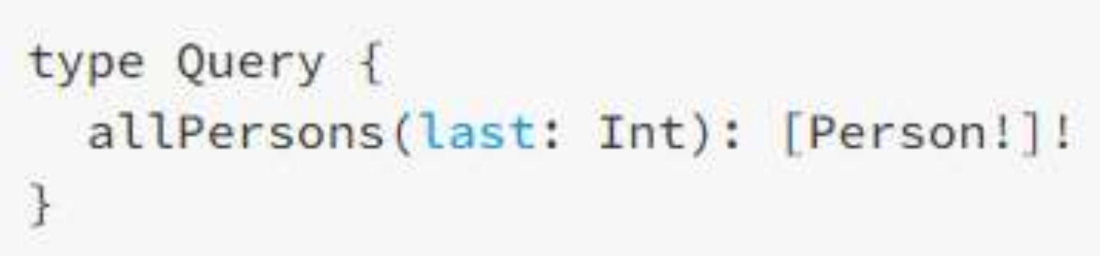
```
type Query { ... }
type Mutation { ... }
type Subscription { ... }
```

LETS REVISIT SCHEMA

- Example of Query with just allpersons
- Example of Query with improved allpersons, where you can specify limit



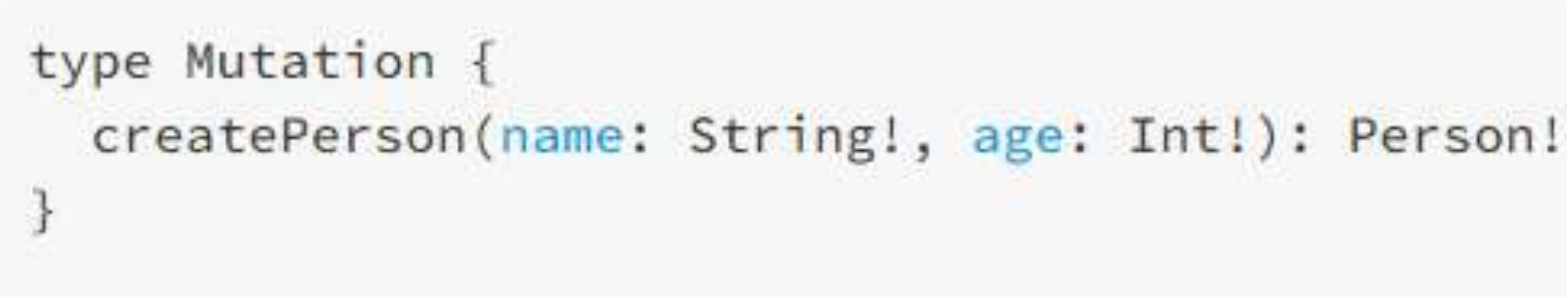
```
type Query {
  allPersons: [Person!]!
}
```



```
type Query {
  allPersons(last: Int): [Person!]!
}
```

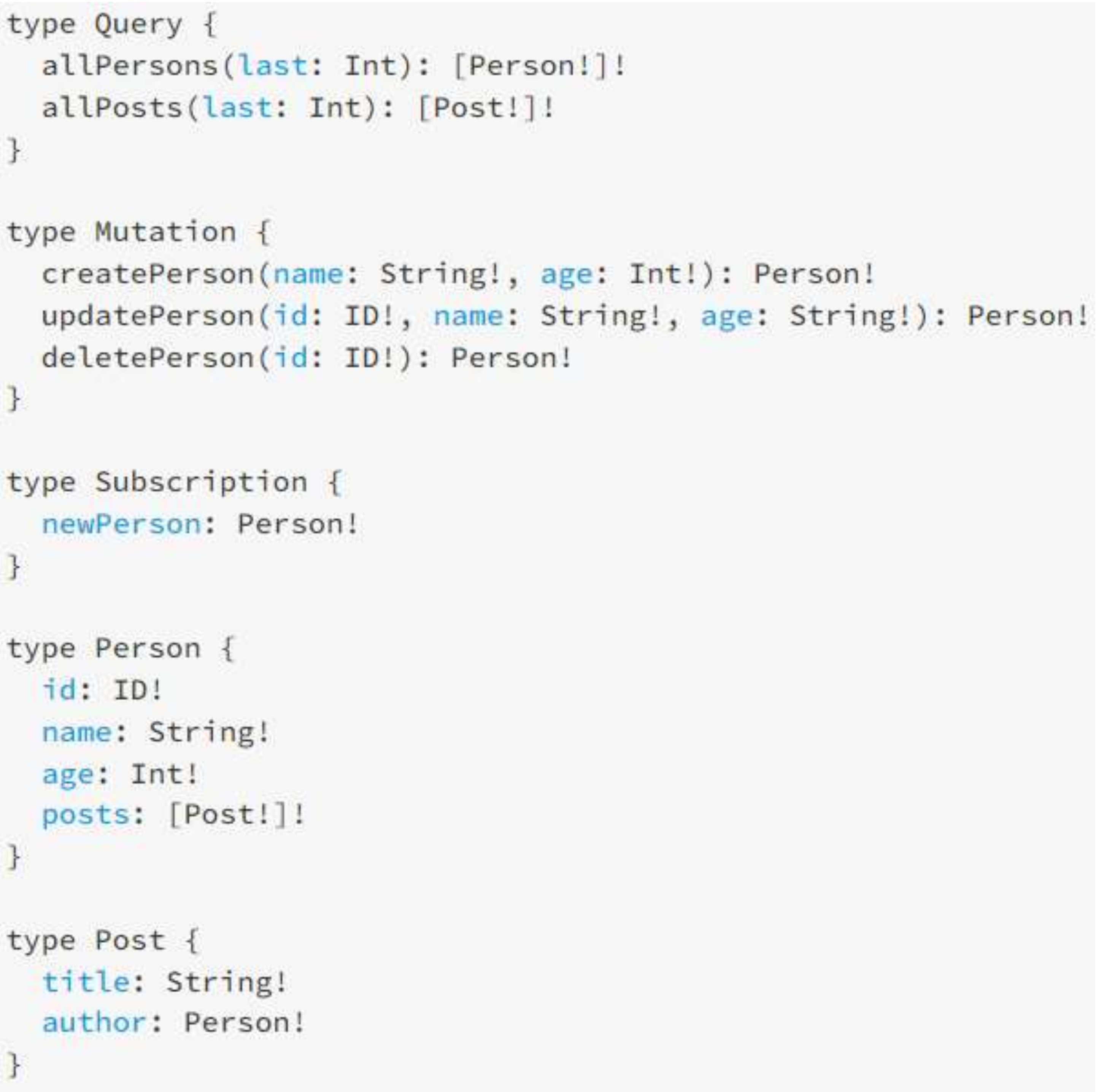
LETS REVISIT SCHEMA

- Example of createperson mutations



```
type Mutation {
  createPerson(name: String!, age: Int!): Person!
}
```

Final Full Schema



```
type Query {
  allPersons(last: Int): [Person!]!
  allPosts(last: Int): [Post!]!
}

type Mutation {
  createPerson(name: String!, age: Int!): Person!
  updatePerson(id: ID!, name: String!, age: String!): Person!
  deletePerson(id: ID!): Person!
}

type Subscription {
  newPerson: Person!
}

type Person {
  id: ID!
  name: String!
  age: Int!
  posts: [Post!]!
}

type Post {
  title: String!
  author: Person!
}
```

REFERENCE

HOW TO GRAPHQL

https://www.howtographql.com/